Object-based SCM: An Efficient Interface for Storage Class Memories

Yangwook Kang   Jingpei Yang   Ethan L. Miller
Storage Systems Research Center, University of California, Santa Cruz
{ywkang, yangjp, elm}@cs.ucsc.edu

Abstract—**Storage Class Memory (SCM) has become increasingly popular in enterprise systems as well as embedded and mobile systems. However, replacing hard drives with SCMs in current storage systems often forces either major changes in file systems or suboptimal performance, because the current block-based interface does not deliver enough information to the device to allow it to optimize data management for specific device characteristics such as the out-of-place update. To alleviate this problem and fully utilize different characteristics of SCMs, we propose the use of an object-based model that provides the hardware and firmware the ability to optimize performance for the underlying implementation, and allows drop-in replacement for devices based on new types of SCM. We discuss the design of object-based SCMs and implement an object-based flash memory prototype. By analyzing different design choices for several subsystems, such as data placement policies and index structures, we show that our object-based model provides comparable performance to other flash file systems while enabling advanced features such as object-level reliability.**

I. Introduction

Storage class memories (SCMs) are playing an increasingly important role in the storage hierarchy. The combination of low power consumption, relatively large capacity, fast random I/O performance, and shock resistance make SCMs attractive for use in desktops and servers as well as in embedded systems. Recently, deployment of Solid State Drives (SSDs) using NAND flash has rapidly accelerated, allowing SCMs to replace disks by providing an interface compatible with current hard drives. Systems using SSDs can deliver much better I/O performance than disk-based systems. However, there are many other SCM technologies beyond NAND flash, including FeRAM, Phase Change RAM (PCM), and carbon nanotube, that may see dramatically increased use in the near future. It is critical to design systems that can both fully utilize flash memory and easily accept drop-in replacements using future technologies.

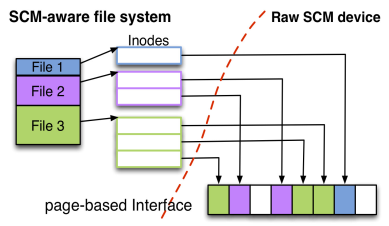
Although SCMs generally provide better performance than hard drives, they require more intelligent algorithms to efficiently handle unique requirements such as out-of-place update and wear-leveling. Because SCM technologies differ in many characteristics, the design of file systems optimized for each technology also varies significantly, creating issues of portability and compatibility. Efforts to exploit these new characteristics in file systems have driven a great deal of research, primarily using one of two approaches; the direct-access model and the FTL-based model. The first model, shown in Figure 1(a), either places SCM on the main memory path and supports direct access from a file system [10], or uses a specific file system and hardware that allows SCMs to work properly in the system [2]. This model provides optimal performance for a specific hardware configuration, but suffers from a potential requirement to redesign the file system to optimally utilize (or simply function properly with) different SCMs or even devices with different configurations of a particular technology.

The second approach, shown in Figure 1(b), interposes firmware (Flash Translation Layer, or FTL) between the raw device and a standard block-based file system, hiding the complexities of managing hardware from the file system and allowing devices to be accessed directly by an unmodified disk file system. However, this approach achieves suboptimal performance due to the lack of file system semantics delivered to the hardware and the duplication of block mapping in both the file system and the device. For example, the use of the TRIM command and *nameless writes* proposed by Arpaci-Dusseau, *et al.* [4] enables SSDs to be more efficient on *delete* and *write* operations. However, existing file systems must be modified to use these operations, and file systems must still maintain their own block maps for files in addition to those kept on the flash device.

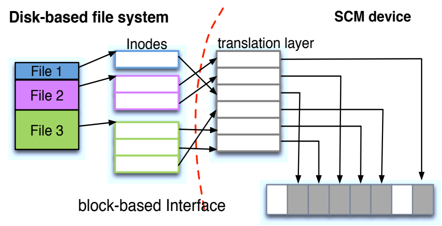
To alleviate the problems of current approaches to integrating SCMs into the file system and exploit the characteristics of various SCM devices without either limiting the design flexibility or introducing additional overhead, we explore the use of an object-based storage model [12, 29] for SCMs. This model offloads the storage management layer from the file system to the underlying hardware, enabling device manufacturers to optimize the mapping layer based on the hardware configuration. The POSIX-level requests are encapsulated in an object with their metadata information and sent to the device through an object-based interface. By doing so, the object-based storage model provides an easy transition between different SCM devices, advanced features such as object-level reliability, and compression. This model can be implemented on any type of SCM device more flexibly, while having one generic object-based file system on the host.

To demonstrate the utility of this approach, we design and implement an object-based model for SCMs based on flash memory, since flash memory appears to have the most

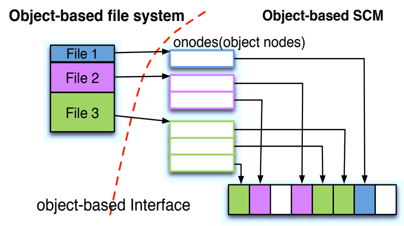
978-1-4577-0428-4/11/$26.00 © 2011 IEEE



(a) Direct access model.



(b) FTL-based model.



(c) Object-based model.

Fig. 1. Three approaches to access SCMs from file system

restrictions among SCMs, and is widely available today. By developing an object-based SCM prototype on flash memory, we show that an object-based interface is both efficient and portable, and show how this prototype can be modified to accommodate other types of SCMs. Based on this model, we also discuss several design choices for the subsystems in object-based SCMs such as data placement policies, index structures and cleaning policies.

The rest of the paper is organized as follows. Section II introduces different SCMs and explains the existing approaches to access SCMs. Section III discusses the benefits of using object-based model for SCMs. Section IV presents our implementation of an object-based model on flash memory, and the effects of several design choices of the subsystems are shown in Section V. Section VI discusses future work and Section VII concludes.

II. Background and Related Work

With the rapid development of semi-conductor technologies, several new types of non-volatile memories have become available. However, the varying characteristics of these memories and the restrictions of the block interface have led to many different storage system designs.

A. *Storage class memories*

Storage class memory is a new class of non-volatile memory devices that includes flash, FeRAM, magnetic RAM [7], phase-change memory (PCM), and carbon nanotube memory [26]. SCM blurs the distinction between main memory and secondary storage devices by providing non-volatility at a cheaper price than DRAM, and faster random access than disk. However, the direct use of SCMs in current systems requires detailed knowledge of the hardware for extracting higher performance and perhaps even proper operation.

Different SCM technologies have different characteristics, as shown in Table I [16]. Flash memory supports page-level access but does not support overwrites due to the need to erase a block before rewriting it. While other types of SCMs such as PCM and MRAM provide byte-addressability and in-place update, each of them is also different in characteristics such as scalability, reliability, wear resistance, performance, and retention—differences that might favor different storage system designs. For example, flash-aware file systems usually use a log-structure to support out-of-place updates, and use a cleaner to reclaim invalidated blocks to generate free blocks for future writing. However, these techniques may

	NAND	PRAM	FeRAM	MRAM
read	15 μs	68 ns	70 ns	35 ns
write	200 μs	180 ns	70 ns	35 ns
erase	2 ms	none	none	none
in-place update	No	Yes	Yes	Yes
cleaner	Yes	No	No	No
write endurance	10^5	10^8	10^{15}	10^{15}
access unit	Page	Byte	Byte	Byte
power consumption	High	High	Low	Low

TABLE I
SCM characteristics. [16]

not be necessary for byte-addressable SCMs. Moreover, other technologies may have their own drawbacks such as byte-level wear-out in PCM and destructive read in FeRAM. Since byte-addressable SCMs use rapidly evolving technologies and there are a number of different candidates, each with different strengths and limitations, it is not feasible to change the host system to fit each change in SCM technology. Thus, both portability and performance are primary issues in designing a SCM-based storage system.

B. *Existing approaches to access SCMs*

Currently, there are two approaches to access SCM devices. These approaches either use a direct access model on the host side (Figure 1(a)) or an FTL-based model embedded in the underlying hardware (Figure 1(b)).

1) Direct access model: The direct access model, shown in Figure 1(a), supports direct access to SCMs by either placing them on the main memory data path, or designing a specific file system that allows SCMs to work properly in the system. For instance, flash-aware file systems such as YAFFS [2], JFFS2 [35], RCFFS [18], and UBIFS [14] directly access flash memory via device drivers. Some storage systems for byte-addressable SCMs place them on the main memory bus, and use them as memory [10, 23], or secondary storage for metadata [11, 21].

The main advantage of this approach is that the core functionalities of the file systems such as indexing and data placement policy can be efficiently implemented for high performance. Since the characteristics of the underlying medium are understood by the file system and the raw medium can be accessed directly, the file system can fully control the medium. However, since both the hardware that supports direct access to SCM and the file system need to be developed, it can be difficult to deploy the system, and migration to new types of SCMs into the running system is problematic. Also, it can be difficult to leverage the features of the underlying hardware because the file system is generally not designed for specific hardware. For example, the file system might be able to optimize I/O operations if it is aware of the existence of multiple data buses to multiple SCM modules in the system.

Portability is another issue with this approach. Since the hardware for accessing SCM is usually integrated into the storage system, it is not easy to move SCMs from one machine to another as is done with hard drives. Moreover, lack of generic interfaces for SCMs and the need for specific hardware for each SCM make this model difficult to adopt in commodity systems.

2) FTL-based model: The second approach, widely used in modern SSD-based systems, is to access SCMs via a translation layer. As depicted in Figure 1(b), the SCM device in this model contains a sector-to-page mapping table, and exposes a block interface to the host, allowing legacy file systems to access the SCM as a block-based storage device. Thus, FTL-based storage devices can replace current hard disks without modifications to the operating system.

However, compatibility with legacy systems comes at a high cost. Since the block interface is designed for disks and supports only read and write sector operations, the device cannot acquire essential information such a the type of "high-level" operation being done and the data type; thus, it cannot leverage this information to better place or reclaim data blocks. As a result, there have been many efforts to improve the performance of the flash translation layer that propose mapping schemes, cache strategies and heuristics [9, 19, 22, 27] to improve these operations. For example, the use of the TRIM command in flash memory allows the device to recognize delete operations so that the device can invalidate data blocks belonging to a deleted file [31]. In spite of these efforts, they are not as efficient as the direct access model, and are often very complex. Moreover, a fundamental problem of the FTL-based model is the existence of two translation layers in the data path, one in the file system, and one in the device.

Two approaches have recently been proposed to overcome the limitations of the FTL-based model by removing or minimizing the mapping table in the device. Arpaci-Dusseau, *et al.* proposed *nameless writes*, allowing the file system to directly manage the mapping table [4]. In this model, the device returns the physical page number to the file system so that it can store a mapping between a logical block number and a physical page number in an inode. DFS is a flash file system that uses a similar approach [15]. It also removes the mapping table from the device, and moves it to the virtualized flash storage layer, which is an FTL that resides in the operating system. Both approaches focus on reducing the complexity of the device and allow the operating system to control the device. However, the nameless writes approach has some limitations in optimizing the flash device: the file system has no information about the underlying medium except for a physical page number, so both the file system and device must still keep a complete block mapping. In DFS, drop-in replacement for other types of SCMs are not easy because they would require a new file system for the underlying SCM or suffer from performance issues of using the block interface.

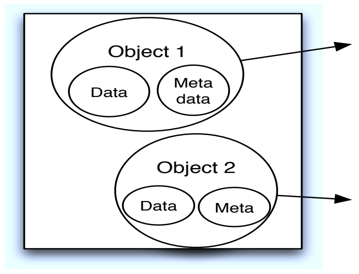
3) Object-based storage model: The object-based storage model has been used in high-performance and large-scale distributed storage systems [8, 12, 33]. These systems, including Panasas [25], Ceph [34], and Slice [3], use object-based storage devices (OSDs) to allow each disk-based device to work independently of the master server, and add more functionality such as replica management, load-balancing, and data placement to the device. Thus, the system can achieve high scalability and robustness, by greatly reducing the neede for central coordination.

While disk-based OSDs provide useful features in distributed systems, they have not gained much attention as a replacement for block based devices because disks work efficiently with the block interface—disks rarely remap blocks. Recently, however, Rajimwale, *et al.* proposed the use of an object-based interface for flash memory [29]. In systems using SCMs where a richer interface than block interface is required, this model could alleviate several problems with existing approaches: the lack of file system level information in the FTL-based model, portability in the direct access model, and efficiency of implementation in both. Under this model, the device can optimize the performance of the device using informed optimizations. Moreover, this approach allows a single file system to support multiple OSDs, making it easy to switch to a new type of OSD or even include multiple types of devices or even technologies in a single file system. More advanced features exploiting the existence of objects such as object-level reliability and compression can also be provided independent of the file system.

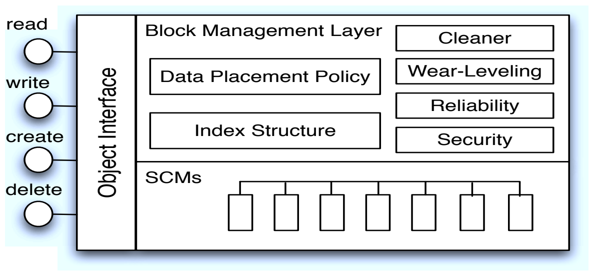
III. Object-based Storage Model for SCMs

The object-based model, shown in Figure 2, consists of two main components: the object-based file system and object-based devices; components communicate using objects. Each object consists of variable-length data and metadata associated with the data such as size and type, allowing the file system to send information about data to the device. An object interface, standardized in ANSI T10 [24], provides operations such as create, delete, read, and write on objects, delivering requests to the device without requiring additional operations such as TRIM.

In systems built on the object-based storage model, a file system does name resolution on the host side, offloading the storage management layer to the OSD. By isolating device-specific technology behind an object-based interface, this model allows the file system to be independent of the particulars of the storage medium while the characteristics of the storage medium are efficiently handled within the device. Thus, a single file system can be efficiently used with different types of SCM devices, in contrast to the current approaches that either require significant changes in the system or sacrifice I/O performance. Since the device manufacturers have better knowledge about the hardware configuration of SCM devices



(a) Object-based file system.



(b) Object-based device.

Fig. 2. System overview of object-based SCMs

than file system designers, this model typically enables better hardware optimizations than native SCM-aware file systems.

The block interface is limited in that it only delivers the sector number and the request type (read or write) to the device. In contrast, the object-based interface delivers objects (which contain both data and associated metadata) and recognizes all types of requests that the file system does. By doing so, the OSD is able to receive the same level of information that the file system maintains, allowing devices to provide features such as hot/cold separation to reduce cleaning overhead that have traditionally been provided by native file systems. For small objects, OSDs can achieve better space efficiency than block-based devices due to the lack of a minimum allocation size. OSDs can reduce index overheads by using extent-based allocation for large, infrequently updated objects. In addition, by encapsulating data in objects, OSDs can provide more advanced features, such as object-level reliability, compression, and execution-in-place. Moreover, adding an object interface will not add significant complexity to existing FTL firmware since SCM devices already need a translation layer for data placement and segment management. For example, when hybrid phase-change/flash memory is used, the file system can store data efficiently by simply sending a write-object request to the OSD; the file system need not know about the two types of memory in the device.

We now discuss the design choices for each subsystem of the object-based SCMs in Section III-A and Section III-B, and we subsequently discuss the advanced features enabled by the object-based model in Section III-C.

A. Object-based File System

An object-based file system maintains files consisting of one or more data objects, which contain the file data, and a single inode object, which records metadata information about the file, such as permission and ownership. However, the inode does not point to the physical blocks, but rather the objects that make up the file, a mapping that can be done algorithmically by having the objects be identified by a combination of the unique file ID and the object's offset in the file.

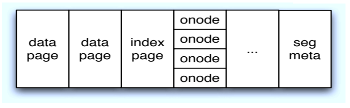
Since objects are identified by a unique identifier, not by human-readable names, there is no information about the logical relationship among objects in this flat file system, offering the file system more flexibility when distributing objects to several OSDs. Replicas of each file can be maintained at the object level by OSDs. Moreover, variable-sized objects can be named with any n-bit identifier that need not have a relationship to the physical location of the data. This sparse name space provides the file system more flexibility when assigning object identifiers for different purposes. Thus, the object-based model can guarantee scalability for SCM-based file systems in large-scale distributed systems [13].

When a file request comes in from the virtual file system (VFS) layer, the file system finds objects related to the request based on an inode number and a file offset, and generates an object request for the data. It then determines whether the requests should be aggregated before being sent to the OSD; since the OSD can deal better with a single large request than with multiple small requests, aggregation can improve performance. For directory operations, the file system can assign an object for each directory entry or store all entries for a directory in a single data object. Although an object-based model uses a flat namespace, support for directories is not very different from that of typical file systems, since typical file systems also need to convert the directory and file name into unique inode numbers in order to maintain a hierarchy. For example, when receiving a `delete` request, the file system only needs to finds the objects that contain the requested range of data, and send one or several `delete_object()` requests to the OSD, instead of generating several `read_block` and `write_block` operations to the device as is done in traditional block-based file systems.

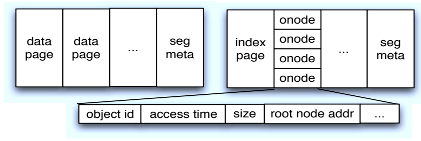
B. Object-based Devices for SCMs

Similar to SSDs, an OSD consists of SCM chips and multiple subsystems required to handle the characteristics of the medium such as a data placement policy and a wear-leveling mechanism, as shown in Figure 1(c). However, the design of its subsystems are more similar to that of typical native block managers for SCMs because OSD subsystems can utilize the same level of information that native file systems have while SSDs require many heuristic algorithms to alleviate the problems of using the block interface, such as a log-block merge optimization.

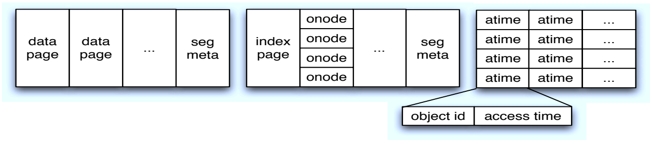
By having rich information about the requests and a block management layer in the device, the OSD approach enables better hardware optimizations as well as simple and efficient subsystems. For example, device manufacturers can split a large write request into small concurrent write requests to multiple SCM chips in order to improve performance. More-



(a) Typical log-structured layout.



(b) Separation of data and metadata.



(c) Separation of data, metadata and access time.

Fig. 3. Three kinds of data allocation policies

over, OSDs can further exploit the existence of objects to provide more advanced features such as object-level reliability and compression. In the following subsections, we discuss the design of core subsystems and new features enabled in the object-based SCMs.

Data placement policies. The focus of a data placement policy is the maximization of performance and lifetime by carefully placing data on SCMs. Typically, flash-based storage systems use a variation of a log-structure to store data because it supports out-of-place update and wear-leveling by placing data and metadata together in a log, an approach we term a *combined* policy, shown in Figure 3(a). Since data and metadata are sequentially written to a log and never updated in this policy, log-structured storage requires a cleaner to generate free space by reclaiming invalidated blocks. Cleaning overhead is known to be a primary factor determining log-structure file system performance [6], so reducing it is a key goal of data placement policies for flash memory. Log structures may also be used for other SCMs which support in-place update to facilitate wear-leveling, so similar design choices could be made in order to reduce the cleaning overhead. The design of data placement policies for byte-addressable SCMs could be changed more flexibly by the manufacturer.

In object-based SCMs, since the type and size of the requests and other hints about data are known, various techniques to reduce the cleaning overhead can be applied. For example, hot-and-cold separation, which stores frequently accessed data blocks in a different segment, can be used since the device can identify which objects or internal data structures are more frequently accessed. Existing intelligent cleaning algorithms that have been developed for native file systems for SCMs can also be adopted with little modification. Our approach to optimize the data placement policy is based on the *combined* policy, with the addition that we separate frequently updated data from cold data such as object metadata and access time. Separating data and metadata was also used in other file systems such as DualFS [28] and hFS [36]. Unlike those systems and other SCM devices that do not manage file metadata internally, this approach can easily be accomplished in object-based SCMs as metadata of objects are maintained by the device itself.

Since the inode no longer maintains the physical addresses of file blocks, the OSD internally maintains the metadata of each object in a data structure called an *onode*. It contains information of each object such as size, access time and object ID as well as a pointer to the object's constituent blocks. An onode is one of the most frequently updated data structures in the device, and thus stored separately in our data placement polices, as shown in Figures 3(b) and 3(c).

Index structures. FTL-based approaches require a complex sector number to page number mapping table to determine the physical addresses of data blocks because the index structure does not recognize which block belongs to which file and the sector number is the only information they can use. Thus, various FTL schemes have been proposed to alleviate this problem. For example, the log-block mapping scheme maintains a small number of blocks in flash memory as temporary storage for overwrites [20]. However, in native file systems for SCMs or OSDs, an index structure utilizes the full semantics of the requests and does not require a complex mapping table, allowing it to be simpler and more efficient.

Since improving space efficiency both reduces the number of I/Os and affects the life time in SCM devices, it is one of the important design issues of an index structure for SCM. For example, YAFFS stores the index in the spare area of each flash page to remove the index overhead [2]. However, it must perform a time-consuming scan of the entire spare area to build an in-memory index structure when mounting. Although YAFFS recently added checkpointing and lazy-loading techniques in order to improve the mounting time, it still requires scanning if the device is not properly unmounted. Thus, as capacity of flash chips grows, this approach would require more time, which can be a tradeoff between index overhead and mounting time.

Another approach to store indices in SCM devices is to use on-media index structures such as an inode-like structure or a variant of a B-tree. UBIFS [14] uses a wandering tree, which is a B^+-tree that supports out-of-place updates by writing all the internal tree nodes whenever a leaf node is updated, resulting in a much higher index overhead than that of a normal B^+-tree. A few attempts have been made to reduce the cleaning overhead of the index structures. For instance, the mu-tree places all the internal nodes in one page so that only one page needs to be written [17]. Subsequently, Agrawal, *et al.* proposed a lazy-adaptive tree, which minimizes the access to the flash medium by using cascaded buffers [1].

In order to reduce the overhead of an index structure in object-based SCMs, we used two optimizations based on a wandering tree with a dedicated cache. The first is the use of extent-based allocation. Since an object has a variable-length, and its size and request type are known to the device, an OSD can efficiently support extents, particularly if the file system can generate large object requests. This tree structure uses a combination of object ID and data type as a key, and a combination of physical address, offset and length as a value. The second approach to reduce the index overhead is to use a write buffer for small objects. Small objects whose size is less than a minimum write unit can be stored together to reduce the internal fragmentation within one unit, thus saving some page I/Os and index records. By sequentially writing large objects, high write performance can be achieved as well.

Wear-leveling and cleaning. Increasing the lifetime of the medium is another important goal in designing SCM devices, since many SCMs burn out after a certain number of writes. Therefore, most SCM devices need to maintain the wear-level of each erase unit and try to use them evenly. Since the storage management layer is inside the device in the object-based storage model, the device manufacturers can freely choose appropriate wear-leveling and cleaning policies for target SCMs.

In SCMs that do not support in-place updates, global wear-leveling is typically done by a log-structure cleaner, which maintains metadata for each segment such as number of erases and age, and selects victim segments based on those values. For byte-addressable SCMs such as PCM, other approaches can be used to track the wear-level of SCMs. Condit, *et al.* proposed two techniques for efficient wear-leveling; rotating bits within a page at the level of the memory controller and swapping virtual-to-physical page mappings [10]. Although some wear-leveling techniques require hardware support, the manufacturers can add more hardware without affecting the host system in the object-based storage model.

In our prototype, different cleaning thresholds are used for different types of segments. Since *atime* (access time) segments do not contain any valid data in them, they are always picked first. We set a lower threshold for metadata segments than data segments because the live data in metadata segments is more likely to be modified soon.

An object-based storage model can keep track of the status of an entire object, so more optimizations are possible. For example, by knowing the update frequency and size of each object, OSDs could place cold objects in blocks where erase counts are higher than others, or group an object's blocks together.

C. Advanced features of Object-based SCMs

Object-level reliability. Most SCM devices used today rely on per-page error correction algorithms, which can detect and correct a certain number of bytes in a page depending on the size of the ECC. However, this approach cannot protect data against whole page failures, reading the wrong data back, or misdirected writes, which store the requested page to the wrong address and return a success code to the file system [5]. An object-based SCM can recover from this type of error by providing per-object parity. The device generates one group of parities for all data pages, and another group of parities for an onode whenever an object is modified. By reading all data blocks belonging to an objects and comparing against the parity, it can detect and correct misdirected writes and whole page failures as well as bit-flips. The amount of parity can be adjusted based on the error rates of the medium by the device manufacturers or specified by the user on a per-object basis. In our prototype, we maintain one parity for data pages and one parity for index nodes for each object.

Object-level compression and encryption. Since OSDs have access to object metadata, they can infer an object's type and determine whether it is a worthwhile candidate for compression. The type of object, such as text or multimedia, can be inferred by reading a hint from a user or by reading the first few bytes of an object. Both compression and encryption can be done by specialized hardware chips in the device to improve I/O performance as well as reducing CPU utilization at the host.

Data compression and encryption can improve space efficiency and security as well as overall performance in SCM devices. In object-based SCMs, the device can provide either system-wide encryption or per-object encryption. The device can determine which object to encrypt based on its content, or users can specify that an object should be encrypted by setting a flag as a hint when sending a request. Thus, the SCM device could set different security levels for different types of objects and do encryption per-object, perhaps using different keys for different objects. Moreover, if the device does both compression and encryption, it can do it in the right order: compression followed by encryption.

Client library for object-based SCMs. A client library can allow users to customize OSD behavior by providing hints to the device. For example, users can provide an encryption key or specify the level of reliability for a specific object. Moreover, the library can provide an interface that bypasses the VFS layer to achieve a smaller memory footprint for small files by avoiding the need to have the VFS assign 4 KB pages to those small files before sending them to the file system. Moreover, an OSD is capable of managing variable-length objects efficiently; in our prototype, we have the file system send a hint to embed inode objects in onodes to reduce index records and page I/Os.

Object-level transaction. It is often difficult to support transactions in block-based devices because the device lacks information about higher-level file system semantics. In object-based storage devices, however, because metadata is handled inside the device and an object can send a hint from the upper layer to the device, the devices can provide transactions without requiring modifications to the host system. For example, users can start the transaction by setting a common transaction ID to objects in the same transaction. The OSD can use this transaction ID to establish a dependency hierarchy and complete the transaction when a top-level object in the hierarchy is being created or deleted. One way to implement the transaction in the device is to use copy-on-write, which writes modifications to a different place until the commit begins, as commonly used in the process subsystem. Moreover, OSDs can seamlessly support per-object transactions, guaranteeing atomic object I/Os.

In summary, the object-based storage model enables device manufacturers to provide SCM devices with various levels of functionality depending on the needs of their customers and make them fully customizable without host system changes. This model has the advantages of both FTL-based storage devices and native file systems: portability and efficiency. It also provides an abstraction of a request that can be used in various subsystems to provide intelligence to the device.

IV. Implementation

We built an object-based model prototype as a file system module in the Linux 2.6 kernel to investigate several design issues in the object-based model. This file system module consists of both the object-based file system and object-based storage device. The two components communicate only via an object interface, and do not share any in-memory data since they need to be independent of each other; they use memory copy instead of passing a pointer.

We picked flash memory as a target SCM for our prototype since it has more restrictions than other types of SCMs and is increasingly used in many devices such as SSDs in spite of several performance issues. The flash memory is simulated using NANDsim, the NAND flash simulator distributed with the kernel. This approach allowed us to use "raw" flash memory for the OSD; commercial flash memory typically contains an FTL and thus does not allow raw access to flash chips.

An object-based file system generates two objects per file; one for file data and one for inode, as shown in Figure 4. They are treated as separate objects in the device; an inode now contains only metadata information that is not frequently changed such as permission and ownership. By using a hint for inode objects, as discussed in Section III-C, inodes are stored in onodes in our prototype.

Support for hierarchical namespaces follows an approach similar to that used in disk-based systems. The directory file contains $\langle objID, dirname \rangle$, so the file system updates a data page in the directory file whenever an entry changes. One optimization we use in this prototype is a hashed directory that contains a pair of an offset of the directory entry and hash of the directory entry name. Given a directory name, it looks up the hash directory to find an offset of the entry, and tries to read the contents of data at the offset. The use of a hash directory helps to reduce lookup time when there are many subdirectories or files under one directory.

Since generating large objects helps reduce the index overhead of the device, an object-based file system maintains a cache for data pages to be written. The data page cache tries to cluster sequential writes to a file and collect them into a few large objects. The maximum size of the cache and each object is configurable to achieve the benefits of clustering multiple small pages into a large object. The object metadata operations, which usually need to be synchronously written, are directly flushed to the device. Notice that the file system does not assume flash memory in this implementation, but rather focuses on object management and the integration with the VFS, and is independent of the implementation of the underlying device.

As with most flash devices, a log-structure is used to handle the no-overwrite restriction and wear-leveling issues. As depicted in Figure 3(a), data and metadata are written sequentially within a segment, and segment metadata, such as timestamp, segment usage, reverse indices and superblock information, is written at the end of each segment. When there are not enough free segments, a cleaner is invoked to reclaim free segments.

Besides a typical log-structure, we implement two new data placement policies, shown in Figures 3(b) and 3(c), that are enabled by the object-based storage model. The idea behind these policies is to reduce the cleaning overhead by separating frequently accessed data from cold data so that the cleaner can pick a segment that has small live data (or metadata) in it. The *split* policy stores metadata and data separately in metadata segments and data segments respectively, based on the assumption that metadata will be more quickly invalidated than data. In this policy, the metadata of an object (its *onode*) and index structure nodes are stored in the metadata segment and other user data is stored in data segments. The *split+atime* policy further separates access time from the rest of the onode, avoiding frequent onode updates due to access time updates when an object is read but not modified. Since the size of an onode is larger than that of an access time entry, this approach can reduce the amount of metadata to written for the read request, thus reducing the cleaning overhead. In this approach, the access times of objects are journaled in the access time segment. Each entry requires only 16 B to store an object ID and access time, so a single 2 KB page can usually hold more than 128 entries, while only 10–20 onodes can be stored in one page. Access time entries are merged to the corresponding onodes when the total number of entries exceeds a certain threshold or when an object with a pending atime update is modified by a write.

Since flash memory does not support in-place updates, we use a wandering tree [14] combined with extent-based allocation in our system to show the effects of extent-based allocation in the object-based model. Each object has its own tree structure (local tree) that contains the physical location of its data blocks, and there is one global tree that maintains the physical location of onodes, small objects and reverse indices, as shown in Figure 4. Small objects—those less than one page long—are maintained by the global tree instead of generating a tree node that has only one entry, thus improving space efficiency.

In order to show the effects of the hints from a file system or user space, we set a flag for an inode so that it can be stored within an onode. Since inodes are very small and onodes need to be read to retrieve the inode object, embedding inodes into onodes can remove at least one page read per each inode operation.

A cleaner is invoked whenever the number of free segments is below a certain value. Unlike a combined policy, which has only one type of segment, victim segments are ordered

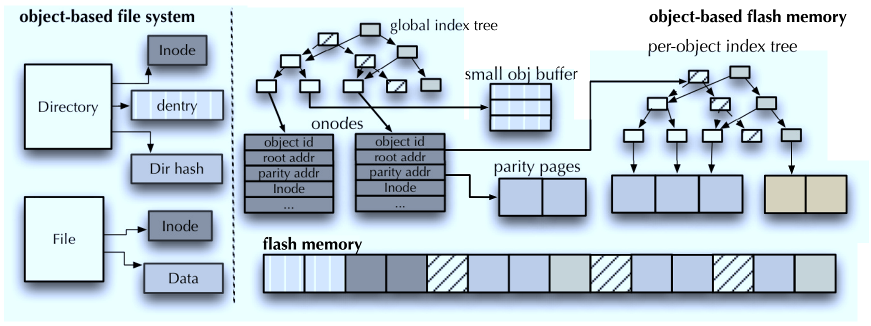


Fig. 4. Implementation of object-based flash memory.

by segment priority. For example, an access time segment contains very little live data, and the data in metadata segments is likely to be invalidated. Thus, we set a different threshold for each type of segment so that access time segments have the highest priority. Data segments have the next highest priority, and metadata segments have the lowest priority. This means a data segment will be picked first if a metadata segment has similar amount of live data, because metadata segments are likely to be invalidated sooner.

To demonstrate the advanced features enabled by the object-based storage model, we implemented object-level reliability in our prototype. For all data pages in an object, the OSD uses a Reed-Solomon code to generate a parity page and algebraic signature to detect bit corruptions [18]. The signature of each data page is stored in the spare area of a flash page, and the parity page is written as a part of an object. Since the algebraic signature and Reed-Solomon code use the same Galois field, the device can run a quick scan to check the validity of an object by the operations of taking the signatures of each page and combining them via XOR; the parity of the signatures of data pages should be the same as the signature of a parity page. For per-object index tree nodes , the file system generates a parity page for tree nodes and stores the page as a part of an object.

There are several other data structures that are adopted to improve the efficiency of the device. For example, since onodes and tree nodes are accessed frequently, the OSD has a cache for each data structure. A one page write buffer is also used to store multiple onodes and small writes in a page. In order to reduce the mount time, the OSD uses the *next-k* algorithm, which enables the file system to quickly find the most recently written segment [18].

The object-based storage devices for byte-addressable SCMs may require different block management policies depending on the characteristics of each type of SCM. For example, the minimum unit of allocation on many SCMs can be adjustable, in contrast to flash memory, which has a fixed page size. A different indexing and data placement scheme can also be used, instead of a log-structure [32]. Although the internal data structure of OSDs might change, the object-based storage model does not require the changes of upper layers in the storage hierarchy. Thus, as SCM technologies evolve, an object-based file system can easily switch to an advanced device without requiring any modification.

V. Evaluation

We evaluate our object-based flash memory using two sets of performance benchmarks. First, we set the Postmark benchmark to generate a read-intensive workload and write-intensive workload, each of which contains a large number of data and metadata operations. These two workloads are used to show the effects of our two data placement policies. The second benchmark we use is a large file benchmark, which Seltzer, *et al.* used to evaluate the log-structured file system [30]. It creates one large file using four types of I/Os; sequential write, sequential read, random write and random read, and used to evaluate extent-based allocation. To measure the cleaning overhead and space efficiency, we make our file system module report the number of segment erases, the number of page I/Os for each subsystem, and the number of bytes copied during cleaning.

Our experiments were conducted on a virtual machine which has a single CPU, 1024 MB RAM, and a 10 GB hard disk. The NANDsim module is configured to model a 128 MB NAND flash memory with 2 KB pages and 128 KB erase blocks. The overall flash size is set by looking at the average response time of `readpage` and `writepage` operations. When we increased the size further, we began suffering from long latency, making the `writepage` operation take up to 4 seconds due to virtual memory reclamation. The use of a small flash size will limit the number of cold data pages we can generate and the size of the segment, making it difficult to look at long-term effects of the cleaning policies. However, each experiment in this section is designed to involve a large number of cleaning operations so that it would not underestimate the cleaning overhead due to the limited flash size; more than 500 segments were cleaned in each experiment. The size of the cache is also set proportional to the size of flash memory: the onode cache is set to 10 KB and the tree node cache to 50 KB. The segment size is set to 256 KB. In the Postmark benchmark, the read/append ratio is set to 1, the smallest number we can set for a write-intensive workload, and the number of files is increased to 1000 while the number of transactions is set to 40000. For a read-intensive workload, we set the read/append ratio to 8 and the number of transactions is increased to 90,000 in order to generate a read-dominant environment. The cleaning threshold for data segments and metadata segments is set to 80% and

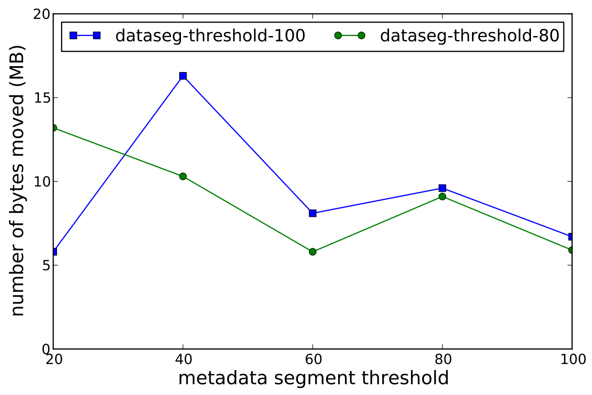


Fig. 5. Effects of cleaning thresholds for different types of segments

60% respectively, based on the result in Section V-B. The default read/append ratio of the Postmark benchmark, which generates five times more metadata than data, is used in other experiments. Each experiment is conducted 20 times and we take an average except for the highest value and the lowest value.

A. Cleaning Overhead of Data Placement Policies

We first measure the cleaning overhead of the three data placement policies under both read-intensive workload and write-intensive workloads using the postmark benchmark. For each policy in Figure 6, the left bar indicates the total number of segments cleaned and the right bar indicates the number of bytes copied during cleaning. Since more onodes and index nodes are written to update access time in the read-intensive workload, both the split policy and the split+atime policy work better than the combined policy, as shown in Figure 6(a). This is because each segment in the combined policy has some invalidated metadata pages and a large number of live data pages, causing only a small amount of free space per victim segment. On the other hand, each metadata segment in the split and split+atime policies contain a very small amount of metadata, thus reducing the cleaning overhead as well as the total number of pages written. The *split+atime* policy further reduces the cleaning overhead because fewer onodes are written by atime journaling.

In the write-intensive workload, separating metadata has a smaller benefit because access times are written together when updating the fields in onodes and these dirty onodes are cached in memory. However, we still get some benefit from the onodes that are read but not written, as shown in Figure 6(b).

B. Effects of cleaning thresholds

Since the three different types of segments have different update frequencies, the device can set a different cleaning threshold for each segment in order to reduce the cleaning overhead. Figure 5 shows the cleaning overhead of the device varying the threshold for data segment and metadata segment. The number after each type of segment represents the percentage of the maximum amount of live data that the victim segments of that type can have. In a pure greedy policy, each

	w/o inode_embed	w/o small obj buf	with all
pages read	378101	389165	364142
pages write	92006	91606	66615
seg write	718	714	288
seg clean	240	235	65

TABLE II
EFFECTS OF INFORMED-PLACEMENT

threshold value is set to 100. Setting a lower threshold for metadata segments works well because the live data in metadata segment is likely to be invalidated quickly, so deferring cleaning often results in metadata segments containing even less live data. However, when the threshold for data segments is less than 60, the flash runs out of space, because there is insufficient space on the flash to have a large number of almost half-empty data segments. The pure greedy policy also works well in this short term cleaning overhead test due to the existence of metadata segments, which contain less live data—the segments that had the least amount of live data were the metadata segments in most cases.

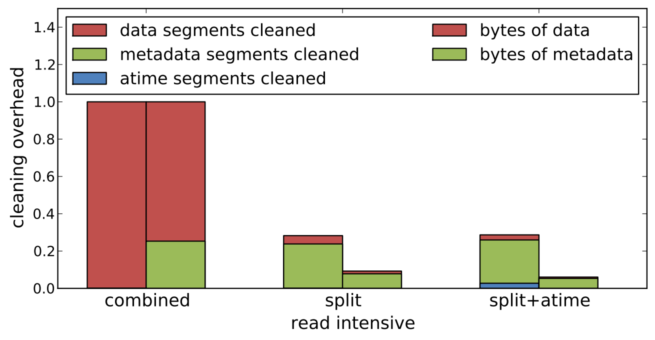
C. Index Structures

The efficiency of the index structure is critical to device performance, especially for devices that do not support in-place update. In order to show the effects of an extent-based allocation, we measure the cleaning overhead and I/O operations with and without extent-based allocation. When extent-based allocation is enabled, physical locations of objects are stored in terms of several *address-length* pairs. Otherwise, each data page is maintained by a wandering tree.

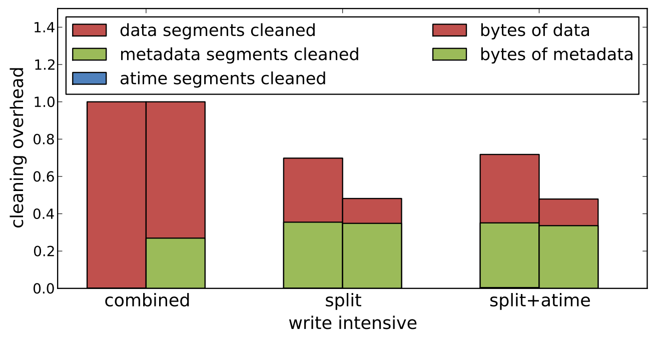
We measure the number of page I/Os issued by the index structure. In the largefile benchmark, since only one large file is sequentially written and then randomly rewritten, the result includes both the benefits of sequential writes and the overhead from random rewrites, and thus it is a good showcase for the effect of an extent-based allocation. Figure 7(a) shows that extent-based allocation significantly reduces both the number of page reads and the number of page writes, regardless of the overhead of random rewrites. In the Postmark benchmark, which generates many small metadata operations and a small number of large data files, the benefit of using extents are minimized, as shown in Figure 7(b). However, even this small file workload still realizes some benefits from extent-based allocation, which reduces the number of page writes by around 1000.

D. Effects of Informed Placement

The file system or user applications can send hints to the device to optimize the behavior of the subsystems on the OSD. For example, the write-amplification problem, which happens when the size of the request is smaller than the minimum unit of writes can be alleviated by using a hint for inode objects, since inode objects are now very small, infrequently updated, and have a fixed size. Because one tree node is generated for each object, the use of this flag reduces the generation of a separate tree node to store the inode and may eliminate some internal tree nodes as well.

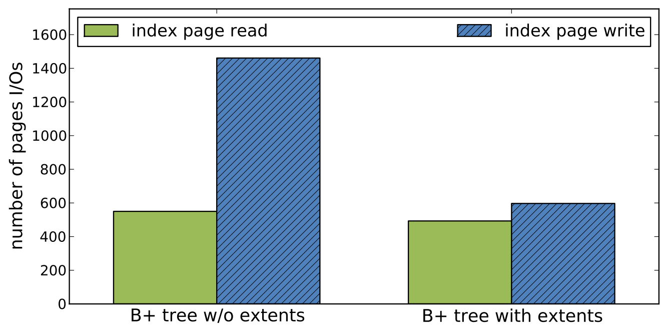


(a) Read-intensive workload

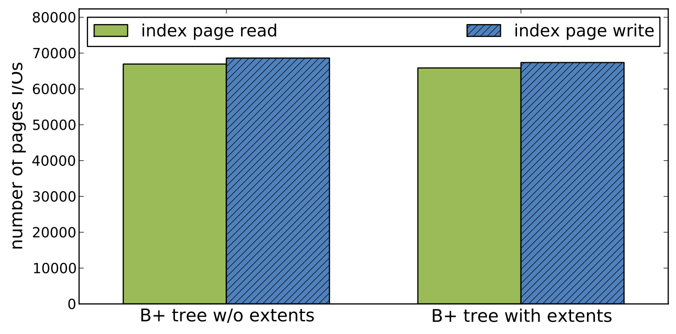


(b) Write-intensive workload

Fig. 6. Cleaning overhead of the three data placement policies. The X-axis represents three data placement policies and the Y-axis is the cleaning overhead normalized to combined policy



(a) Largefile benchmark



(b) Postmark benchmark

Fig. 7. Effects of an extent-based allocation

The size of an object can also be used as a hint for the device. We use a one page buffer for small objects, so the device can store multiple small objects in a single page. In our implementation, a directory entry fits in this category. The size of each directory entry is around 256 B; thus, the OSD could store around 10 entries per 2 KB page.

To show the effects of these optimizations, we set the Postmark benchmark to create more files by setting the number of files is to 2000 and executing 30,000 transactions including `read`, `append`, and `delete` operations. The read/append ratio is set to 5. As shown in Table II, when inode objects are not stored with onodes, they consume more pages, thus increasing the cleaning overhead due largely to the additional index overhead. If a small object buffer is not enabled, each directory entry has to use one flash page due to write-amplification, thus increasing the number of page I/Os. With both optimizations enabled, the number of page I/Os is reduced by more than 65%, and the overall cleaning overhead is also significantly reduced.

E. *Object-level Reliability*

The use of object-level reliability allows the device to detect more types of errors than the current error correction mechanism, which stores 4–8 bytes of ECC on the spare area of each page. It can even be used in conjunction with the current error correction mechanism to reduce the recovery time for simple errors such as a single bit-flip. In this section, we measure how much additional space is required in order to support object-level reliability. Since we generate two parity pages per each object, and the parities are updated whenever an object needs a modification, the overhead is proportional to the number of files and the number of operations.

We separately measure the parity overhead—the number of pages written by the object-level reliability mechanism—and compare it with the total number of page I/Os during the benchmark while increasing the number of files from 500 to 1000 with the same transaction size. As shown in Table III, the number of pages written or read by the object-level reliability mechanism increases, but the overall overhead is less than 10% of the total I/Os, and there is no significant performance difference between the two setups. Object-level reliability thus provides detection and correction of errors that cannot otherwise be achieved while incurring a minimal performance overhead,

F. *Overall Performance*

Lastly, we compare the overall performance of our file system module with the currently available flash-aware file systems: YAFFS, JFFS2, and UBIFS. Neither YAFFS nor JFFS2 has on-media index structures, so they have less index overhead, but both, as a result, require more mounting time, especially when the file system is not unmounted cleanly. UBIFS uses write-back, compression, and a small write buffer to improve both performance and space-efficiency. In our file system, we use the split+atime policy, B^+ tree combined

number of files	total page read	total page write	parity read	parity write	with obj_reliab (sec)	w/o obj_reliab (sec)
500	93633	43822	2437	3388	16.50	15.25
750	140326	86740	5666	7865	21.67	20.25
1000	273931	171369	14892	33827	24.50	23.00

TABLE III
SPACE AND PERFORMANCE OVERHEAD OF OBJECT-LEVEL RELIABILITY.

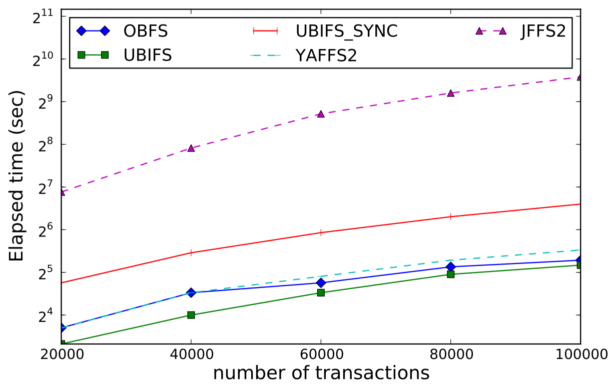


Fig. 8. Overall performance

with extent-based allocation, small object buffers and inode embedding.

Figure 8 shows the overall performance of flash-aware file systems and our file system under a read-intensive workload and write-intensive workload. OBFS represents our prototype, the object-based SCM for flash memory, and UBIFS_SYNC is the ubifs file system runs in synchronous mode. UBIFS shows the best performance among the file systems as it uses write-back while other file systems are using write-through. When the write-back is turned off, the performance became slower than YAFFS and OBFS.

Overall, OBFS shows performance comparable to other flash-aware file systems, even though the block management layer resides in the device and each file system uses different optimizations. For example, YAFFS and JFFS2 have less index overhead and UBIFS uses compression to increase the space-efficiency. Our prototype can be further improved by optimizing the implementation and adopting other features such as compression and write-back. Moreover, object-based SCMs can further improve performance by hardware optimizations in the real devices.

VI. Future work

We are exploring ways of using byte-addressable SCMs as a hybrid with flash memory or as the main storage medium in an object-based storage model. One of the primary issues we are considering is reliability. Since power-cycling does not recover the system from failure in those devices, we need to rollback unnecessary changes in the media and also provide data integrity. Wear-leveling is still important in these SCMs because they could be used as a main memory, which could possibly have lots of small random updates, and tracking the wear-level of each byte is not realistic. There are several other optimizations that could improve the performance of an object-based flash device, such as write-back and compression. Although the cleaning overhead heavily depends on the cleaning policy, only the greedy policy and high-threshold policy are explored in this paper. We plan to implement other cleaning policies to measure long-term wear-leveling effects and cleaning overhead. The implementation can be further optimized by revisiting the use of locks.

We are also exploring techniques to better integrate the object-based interface for SCMs with other operating system services such as execution and transaction support. By providing a slightly richer interface, we can enable powerful yet flexible integration of SCMs into operating systems without requiring wholesale changes each time SCM technology changes.

VII. Conclusion

As storage class memories become popular in storage systems, the need to efficiently use these devices increases. To avoid the limitations of standard block-based interfaces, we propose the use of object-based SCMs, which provide portability and efficiency by delegating storage management to SCM devices and eliminating duplicate mapping tables on both host and device. Further, this approach allows systems to immediately utilize new SCM technologies with no change to host systems, providing flexibility to system designers without the need to optimize the file system for many different types of SCMs. Moreover, this approach can also provide new functionality such as object-level reliability and object-level compression by leveraging the semantics of object-based requests.

By implementing our object-based model prototype for flash memory in the 2.6 Linux kernel, we explored several design issues in object-based SCMs: three data placement policies, two index structures, and other possible optimizations enabled in the object-based storage model such as object-level reliability and embedding inodes into onodes. Our experiments show that separating frequently accessed metadata from data can reduce the cleaning overhead in both workloads, and a B^{+} tree variant with an extent-based allocation can further reduce overhead. We also showed that object-level reliability can be added to the existing reliability mechanism improving the recoverability without incurring much overhead. The performance of our object-based model prototype is comparable with other flash-aware file systems, which are much more efficient than current SSD-disk file system pairings. By using an object-based model for storage class memories, systems can realize the full performance and reliability benefits of current flash-based SCM, while avoiding the need to rewrite the file system as new SCM technologies are deployed.

ACKNOWLEDGMENTS

This research was supported in part by the National Science Foundation under award CCF-0937938, the Department of Energy's Petascale Data Storage Institute under award DE-FC02-06ER25768, and the Center for Research in Intelligent Storage (CRIS) under NSF grant IIP-0934401. Support was also provided by the industrial sponsors of CRIS and the Storage Systems Research Center (SSRC), including IBM, LSI, NetApp, Northrop Grumman, and Samsung Information Systems America.

REFERENCES

[1] D. Agrawal, D. Ganesan, R. Sitaraman, Y. Diao, and S. Singh. Lazy-adaptive tree: an optimized index structure for flash devices. *Proc. VLDB Endow.*, 2:361–372, August 2009.

[2] Aleph One Ltd. YAFFS: Yet another flash file system. http://www.yaffs.net.

[3] D. C. Anderson, J. S. Chase, and A. M. Vahdat. Interposed request routing for scalable network storage. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation (OSDI)*, Oct. 2000.

[4] A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and V. Prabhakaran. Removing the costs of indirection in flash-based SSDs with nameless writes. In *Proceedings of the 2nd Workshop on Hot Topics in Storage and File Systems (HotStorage '10)*, June 2010.

[5] L. N. Bairavasundaram, G. R. Goodson, B. Schroeder, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. An analysis of data corruption in the storage stack. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies (FAST)*, pages 223–238, Feb. 2008.

[6] T. Blackwell, J. Harris, , and M. Seltzer. Heuristic cleaning algorithms in log-structured file systems. In *Proceedings of the Winter 1995 USENIX Technical Conference*, pages 277–288. USENIX, Jan. 1995.

[7] H. Boeve, C. Bruynseraede, J. Das, K. Dessein, G. Borghs, J. De Boeck, R. C. Sousa, L. V. Melo, and P. P. Freitas. Technology assessment for the implementation of magnetoresistive elements with semiconductor components in magnetic random access memory (MRAM) architectures. *IEEE Transactions on Magnetics*, 35(5):2820–2825, Sept. 1999.

[8] L.-F. Cabrera and D. D. E. Long. Swift: Using distributed disk striping to provide high I/O data rates. *Computing Systems*, 4(4):405–436, 1991.

[9] H. J. Choi, S.-H. Lim, , and K. H. Park. JFTL: A flash translation layer based on a journal remapping for flash memory. *ACM Trans on Storage*, 4(4), Jan. 2009.

[10] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. Lee, D. Burger, and D. Coetzee. Better I/O through byte-addressable, persistent memory. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP '09)*, pages 133–146, Oct. 2009.

[11] I. H. Doh, J. Choi, D. Lee, and S. H. Noh. Exploiting non-volatile RAM to enhance flash file system performance. In *7th ACM & IEEE Conference on Embedded Software (EMSOFT '07)*, pages 164–173, 2007.

[12] G. A. Gibson and R. Van Meter. Network attached storage architecture. *Communications of the ACM*, 43(11):37–45, 2000.

[13] J. He, A. Jagatheesan, S. Gupta, J. Bennett, and A. Snavely. DASH: a recipe for a Flash-based data intensive supercomputer. In *Proceedings of SC10*, 2010.

[14] A. Hunter. A brief introduction to the design of UBIFS. http://www.linux-mtd.infradead.org/doc/ubifs_whitepaper.pdf.

[15] W. K. Josephson, L. A. Bongo, K. Li, and D. Flynn. DFS: A file system for virtualized flash storage. *ACM Transactions on Storage*, 6(3), Sept. 2010.

[16] J. Jung, Y. Won, E. Kim, H. Shin, and B. Jeon. FRASH: Exploiting storage class memory in hybrid file system for hierarchical storage. *ACM Transactions on Storage*, 6(1):1–25, 2010.

[17] D. Kang, D. Jung, J.-U. Kang, and J.-S. Kim. μ-Tree : An ordered index structure for nand flash memory. In *7th ACM & IEEE Conference on Embedded Software (EMSOFT '07)*, pages 144–153, 2007.

[18] Y. Kang and E. L. Miller. Adding aggressive error correction to a high-performance compressing flash file system. In *9th ACM & IEEE Conference on Embedded Software (EMSOFT '09)*, Oct. 2009.

[19] H. Kim and S. Ahn. BPLRU: a buffer management scheme for improving random writes in flash storage. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies (FAST)*, 2008.

[20] J. Kim, J. M. Kim, S. H. Noh, S. L. Min, and Y. Cho. A space-efficient flash translation layer for CompactFlash systems. *IEEE Transactions on Consumer Electronics*, 48(2):366–375, May 2002.

[21] J. K. Kim, H. G. Lee, S. Choi, and K. I. Bahng. A PRAM and NAND flash hybrid architecture for high-performance embedded storage subsystems. In *8th ACM & IEEE Conference on Embedded Software (EMSOFT '08)*, pages 31–40, 2008.

[22] S.-W. Lee, D.-J. Park, T.-S. Chung, D.-H. Lee, S. Park, and H.-J. Song. A log buffer-based flash translation layer using fully-associative sector translation. *ACM Trans. on Embedded Computing Systems*, 6(3), 2007.

[23] J. C. Mogul, E. Argollo, M. Shah, and P. Faraboschi. Operating system support for NVM+DRAM hybrid main memory. In *Proceedings of the 13th Workshop on Hot Topics in Operating Systems (HotOS-XIII)*, 2009.

[24] D. Nagle, M. E. Factor, S. Iren, D. Naor, E. Riedel, O. Rodeh, and J. Satran. The ANSI T10 object-based storage standard and current implementations. *IBM Journal of Research and Development*, 52(4):401–411, 2008.

[25] D. Nagle, D. Serenyi, and A. Matthews. The Panasas ActiveScale storage cluster—delivering scalable high bandwidth storage. In *Proceedings of the 2004 ACM/IEEE Conference on Supercomputing (SC '04)*, Nov. 2004.

[26] Nantero, Inc. Nano-ram. http://www.nantero.com/.

[27] S.-Y. Park, D. Jung, J.-U. Kang, J.-S. Kim, and J. Lee. CFLRU: a replacement algorithm for flash memory. In *Proc. of the 2006 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, pages 234–241, 2006.

[28] J. Piernas, T. Cortes, and J. M. García. DualFS: a new journaling file system without meta-data duplication. In *Proceedings of the 16th International Conference on Supercomputing*, pages 84–95, New York, NY, 2002.

[29] A. Rajimwale, V. Prabhakaran, and J. D. Davis. Block management in solid-state devices. In *Proceedings of the 2009 USENIX Annual Technical Conference*, June 2009.

[30] M. Seltzer, K. A. Smith, H. Balakrishnan, J. Chang, S. McMains, and V. Padmanabhan. File system logging versus clustering: A performance comparison. In *Proceedings of the Winter 1995 USENIX Technical Conference*, pages 249–264, 1995.

[31] F. Shu and N. Obr. Data set management commands proposal for ATA8-ACS2. http://t13.org/Documents/UploadedDocuments/docs2008/e07154r6-Data_Set_Management_Proposal_for_ATAACS2.doc.

[32] S. Venkataraman, N. Tolia, P. Ranganathan, and R. H. Campbell. Consistent and durable data structures for non-volatile byte-addressable memory. In *Proceedings of the 9th USENIX Conference on File and Storage Technologies (FAST)*, 2011.

[33] F. Wang, S. A. Brandt, E. L. Miller, and D. D. E. Long. OBFS: A file system for object-based storage devices. In *Proceedings of the 21st IEEE / 12th NASA Goddard Conference on Mass Storage Systems and Technologies*, pages 283–300, College Park, MD, Apr. 2004. IEEE.

[34] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI)*, Nov. 2006.

[35] D. Woodhouse. The journalling flash file system. In *Ottawa Linux Symposium*, July 2001.

[36] Z. Zhang and K. Ghose. hFS: A hybrid file system prototype for improving small file and metadata performance. In *Proceedings of EuroSys 2007*, Mar. 2007.